Static Analysis of Model Transformations for Effective Test Generation

Jean-Marie Mottu, Sagar Sen, Massimo Tisi, Jordi Cabot

Static Analysis of Model Transformations for Effective Test Generation

Jean-Marie Mottu
Université de Nantes - LINA (UMR CNRS 6241)
Nantes, France
Email: jean-marie.mottu@univ-nantes.fr

Sagar Sen, Massimo Tisi, Jordi Cabot
École des Mines de Nantes - INRIA, LINA
Nantes, France
Email: {sagar.sen,massimo.tisi,jordi.cabot}@mines-nantes.fr

***Abstract*—Model transformations are an integral part of several computing systems that manipulate *interconnected graphs of objects called models* in an input domain specified by a metamodel and a set of invariants. Test models are used to look for faults in a transformation. A test model contains a specific set of objects, their interconnections and values for their attributes. Can we automatically generate an effective set of test models using knowledge from the transformation? We present a white-box testing approach that uses static analysis to guide the automatic generation of test inputs for transformations. Our static analysis uncovers knowledge about how the input model elements are accessed by transformation operations. This information is called the input metamodel *footprint* due to the transformation. We transform footprint, input metamodel, its invariants, and transformation pre-conditions to a constraint satisfaction problem in Alloy. We solve the problem to generate sets of test models containing traces of the footprint. Are these test models effective? With the help of a case study transformation we evaluate the effectiveness of these test inputs. We use mutation analysis to show that the test models generated from footprints are more effective (97.62% avg. mutation score) in detecting faults than previously developed approaches based on input domain coverage criteria (89.9% avg.) and unguided generation (70.1% avg.).**

***Keywords*-White Box Testing; Model-Driven Engineering; Model Transformation Testing; Alloy; Automatic Model Completion; Mutation Analysis**

I. Introduction

Model-driven applications are based on interconnected graphs of objects, i.e. *models*, to represent a number of complex software artifacts at development-time and/or runtime, such as source code and structured data. Frameworks for model-driven applications like the Eclipse Modeling Framework EMF[1] are extensively used in a wide range of scenarios[2] and even the highly anticipated Eclipse 4 platform is being developed using EMF[3].

Model transformations are core components that automate model manipulation steps in these systems, such as model refinement, re-factoring for model improvement, aspect weaving, and code generation. Testing model transformations is a relatively new area of software reliability and it presents several new challenges [3]. Model transformation testing requires the specification of software artifacts called *test models* that aim to detect faults in the system. Specifying test models manually is a tedious task, complicated by the fact that they must conform to complex modelling language specifications and constraints. The issue becomes crucial when a tester needs to create several hundreds of test models, that focus on different testing objectives.

Generating large sets of effective test models is the global subject of our work. In [27], we propose a methodology and tool PRAMANA to *automate the generation of test models* while combining heterogenous sources of knowledge. The approach is based on transforming heterogeneous sources of knowledge, such as the *input domain specification* of a transformation given by an input metamodel, well-formedness rules, pre-conditions, and model fragments [12] to a constraint satisfaction problem in ALLOY [18][17]. Solving the ALLOY model and transforming the solutions to instances of the input metamodel gives us a set of test models. The tool is extensible and allows inclusion of new testing strategies. Based on this underlying framework, in [28], the authors take a step further and generate thousands of models by introducing input domain partitioning strategies [12]. We show via mutation analysis [25] that these test models could detect an average of 82% of the faults injected in the model transformation. Nevertheless, the partitioning strategies are essentially *black-box* and have limited effectiveness not using knowledge *from* the transformation.

Recently, in [31], we perform an empirical study to demonstrate that *incomplete partial knowledge* about potential faults in transformations can be used to generate complete test models. These completed test models detect the same faults that fully human-made models satisfying several well-formedness rules can detect. However, both completed test models and human-made models contain testing knowledge obtained from *manual analysis of potential faults* in a transformation. In this paper, we go a step further and ask, *can we automatically extract partial knowledge from model transformations to generate effective test models?* In other words, we are interested in an *automated white-box testing approach* for model transformations. We address three principal challenges to answer this question:

Challenge 1: How to extract partial testing knowledge from

[1] http://www.eclipse.org/modeling/emf/

[2] http://wikipedia.org/wiki/List_of_Eclipse_Modeling_Framework_based_software

[3] http://www.eclipse.org/e4/resources/e4-whitepaper.php

a model transformation?
Challenge 2: How can we automatically generate complete test models that use partial knowledge from a transformation?
Challenge 3: Are automatically completed models from transformation knowledge effective in detecting faults in the transformation?

We present an integrated *white-box methodology* to generate test models from partial knowledge extracted via static analysis of a model transformation.

First we address **Challenge 1**, by extracting partial knowledge from the model transformation about its usage of the input metamodel. We statically analyze a transformation to identify classes and properties used by each transformation operation. We represent this information in what we call a *metamodel footprint* [19]. We present a *footprint strategy* that *partitions the domain of footprint properties* and combines them to generate *model fragments* containing footprints.

To address **Challenge 2** we transform the model fragments to a set of constraints expressed as ALLOY *predicates*. These are the predicates that must be satisfied by the test models we want to generate. We also semi-automatically transform the input domain specification consisting of an input metamodel, well-formedness invariants, pre-conditions to an ALLOY *domain specification* using our tool PRAMANA [27]. The ALLOY specification is simply juxtaposed with the ALLOY *footprint predicates*. This is how we introduce our new forms of testing knowledge. An ALLOY SAT solver generates the test models from the ALLOY specification. The ALLOY SAT solver is parametrized with a *generation design*, that is made of parameters defining the range of the integer values, the number of non-isomorphic test models per predicate, and the scope/bound on each input metamodel element expected in a test model.

Are these test models from ALLOY footprint predicates effective in detecting faults in a transformation? This is a question we address experimentally for **Challenge 3**. We use *mutation analysis* [9] for model transformations [25] to experimentally evaluate the fault-detecting effectiveness of the set of test models from footprints. In our experiments, we employ the representative case study of transforming simplified UML class diagram models to database (RDBMS) models called class2rdbms. Mutation analysis on this transformation reveals that *footprints give highly efficient test models* that detect an average of 97.62% of all injected faults. This mutation score is better than our previous results, 82% average [28] (improved to 89.9% in the experiments of this paper), obtained using input domain coverage strategies (that are transformation-independent). The result suggests that automatic metamodel footprints are an effective source of knowledge to detect faults in transformations. They are almost as effective as hand-made partial models in [31] that can detect all faults when completed.

We summarize the contributions of the paper as follows:
Contribution 1: A white-box strategy based on metamodel footprints to acquire effective knowledge from model transformations to test them. The strategy is integrated in an existing test model generation tool PRAMANA.
Contribution 2: An experimental validation based on mutation analysis to demonstrate that the footprint strategy, a white-box testing strategy, gives better results than well-known black-box strategies based on unguided generation or input domain partitioning.

The paper is organized as follows. In Section II we present the problem description and the representative case study for white-box transformation testing. Sections III, IV and V address our three challenges. In Section III, we present the footprint strategy for the extraction of testing knowledge, in the form of footprint model fragments, from a model transformation. In Section IV, we present an integrated methodology to detect inconsistent footprint predicates and generate test models satisfying valid ones. In Section V, we use mutation analysis in experiments to validate the effectiveness of the generated test models. In Section VI we discuss related work. We conclude in Section VII.

II. PROBLEM DESCRIPTION

Model transformations are expressed in transformation languages that are proficient in navigating, querying, creating, analyzing and modifying models. There exist many paradigms of transformation languages [8] such as imperative (e.g., Kermeta), rule-based (e.g., ATL [21]), and based on graph-transformation (e.g., VIATRA [33]). Automatically testing a model transformation consists in the generation of test models that conform to the input domain of the transformation and contain knowledge that could uncover faults in the transformation code.

In this paper, we perform white-box testing of transformations: the knowledge to generate test models is extracted analyzing the model transformation code. While in the experimentation we consider transformations written in the imperative language Kermeta, the application of our methodology and tools to other transformation languages is straightforward: only the initial phase needs a language-dependent extension.

In Section II-A, we define the problem of white-box testing and introduce the required notation. In Section II-B, we present a representative case study transformation.

A. Automated White-box Transformation Testing: The Problem

The problem of generating test models from white-box knowledge is illustrated in Figure 1. We test the model transformation MT. The input domain of MT is specified by the input metamodel MM_I. The input metamodel MM_I is constrained by a set of invariants $Inv(MM_I)$ that specify well-formedness rules for the input models. The transformation MT has a set of pre-conditions MT in the set $pre(MT)$ that further constrains MM_I defining the specific subset of models that can be processed by MT. These heterogenous sources of knowledge may be expressed in different modelling languages or formats. For instance, MM_I is often expressed

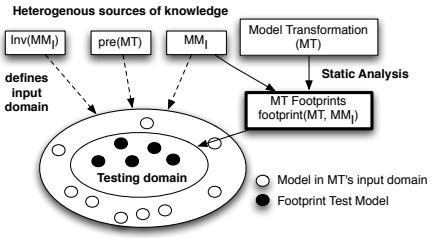


Fig. 1. White-box model transformation testing using footprints

as an EMF Ecore metamodel [7]. Invariants $Inv(MM_I)$ and $pre(MT)$ are expressed in Object Constraint Language (OCL) [26]. The OCL and Ecore are industry standards used to develop metamodels and specify arbitrary invariants on them.

A static analysis of MT can reveal which elements in the input metamodel MM_I are used by MT operations. This information is called metamodel footprint. A footprint $footprint(MT, MM_I)$ is a set of 3-tuples $\langle Operation, Feature, Type \rangle$ where $Operation$ is the name of an operation in MT, $Feature$ is a metamodel element or property used by an expression in $Operation$, and $Type$ is the type of the metamodel element $Feature$. The problem is: how can we use the heterogenous sources of knowledge given in the 5-tuple, $\langle MM_I, Inv(MM_I), pre(MT), footprint(MT, MM_I) \rangle$ to automatically generate a set of test models?

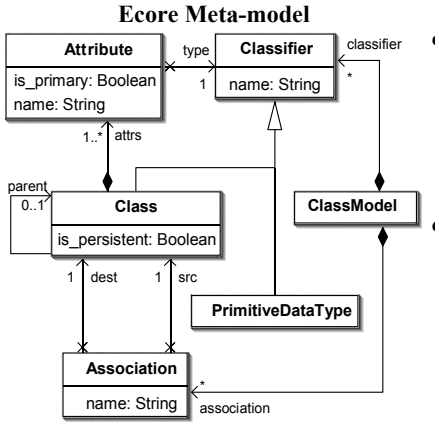
B. Case Study

We address the problem using a case study transformation from simplified UML Class Diagram models to RDBMS models called class2rdbms. The transformation is the benchmark proposed in the MTIP workshop at the MoDELS 2005 conference [4] to experiment and validate model transformation language features, and since then it has been used in several works.

In Figure 2, we present the simplified UMLCD input metamodel for class2rdbms. The concepts and relationships in the input metamodel are stored as an Ecore model [7] (Figure 2(a)). Part of all the invariants $Inv(MM_I)$ on the simplified UMLCD Ecore model, expressed in Object Constraint Language (OCL) [26], are shown in Figure 2(b). The Ecore model and the invariants together represent the true input domain for class2rdbms.

In this paper, we use the Kermeta implementation of class2rdbms, provided in [25]. Constraints in the pre-condition for class2rdbms include: (a) All Class objects must have at least one primary Attribute object (b) A Class object cannot have an Association and an Attribute object of the same name (c) There are no association cycles between non-persistent Class objects.

The transformation class2rdbms serves as a sufficient case study for several reasons. The input domain metamodel of simplified UMLCD covers all major metamodelling concepts



(a)

OCL Invariants

context Class

inv noCyclicInheritance:
not self.allParents()->includes(self)

inv uniqueAttributesName:
self.attrs->forAll(att1, att2 |
att1.name=att2.name implies att1=att2)

context ClassModel

inv uniqueClassifierNames:
self.classifier->forAll(c1, c2 |
c1.name=c2.name implies c1=c2)

inv uniqueClassAssociationSourceName :
self.association->forAll(ass1, ass2 |
ass1.name=ass2.name implies
(ass1=ass2 or ass1.src != ass2.src))

(b)

Fig. 2. (a) Simplified UML Class Diagram Ecore metamodel (b) OCL constraints on the metamodel

such as inheritance, composition, finite and infinite multiplicities. The constraints on the simplified UMLCD metamodel contain both first-order and higher-order constraints. There also exists a constraint to test transitive closure properties on the input model, e.g., there must be no cyclic inheritance. The transformation exercises most major model transformation operations such as navigation, creation, and filtering (described in more detail in [25]) enabling us to test essential model transformation features.

III. Extracting Testing Knowledge from Model Transformations

Generating effective test models (that can detect faults) requires testing knowledge from an intelligent source. We present an automated approach to extract *testing knowledge* from the *model transformation under test* by statically analyzing it. This approach is used in our overall methodology to generate test models using our tool PRAMANA (as described in Section IV).

We illustrate the process of extracting testing knowledge in Figure 3. We describe the two steps of the process in the following sub-sections.

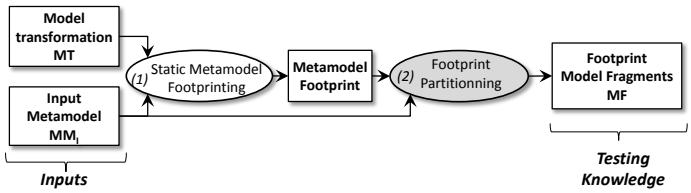


Fig. 3. Extraction of partial test knowledge

1) Static Metamodel Footprinting

The first step is the static analysis of a model transformation called *static metamodel footprinting*. This analysis consists of collecting metamodel elements referenced by transformation operations. In an imperative model transformation language such as Kermeta [24], metamodel elements are collected along the *control flow graph* of the transformation. For every operation in the model transformation we create tuples $\langle Operation, Feature, Type \rangle$ with the name of the operation, the feature of the metamodel in use, and the type of the feature. A detailed explanation of the footprinting process is described in Jeanneret et. al. [19]. The implementation of static footprinting in Kermeta [24] is available in a technical report [20]. This transformation is depicted as step (1) of Figure 3.

The class2rdbms case study gives us 43 *footprint tuples*. Table I shows the footprint tuples generated for two operations of class2rdbms: getAllClasses and getPersistentClass. The operation getAllClasses contains expressions using the class Classifier, the class ClassModel, and the property ClassModel::classifier of type Classifier. The transformation operation getPersistentClass involves the class Class, the relation Class::is_persistent of type Boolean, and the relation Class::parent of type Class. The footprint tuples will be used as a source of structural information to eventually generate test models.

TABLE I
A SUBSET OF METAMODEL FOOTPRINT TUPLES FROM CLASS2RDBMS

Operation	Metamodel Feature	Types
getAllClasses	Classifier	Classifier
getAllClasses	ClassModel	ClassModel
getAllClasses	ClassModel::classifier	Classifier
getPersistentClass	Class	Class
getPersistentClass	Class::is_persistent	Boolean
getPersistentClass	Class::parent	Class

2) Transformation from Footprint to Model Fragments

In this paper we propose a *novel strategy to generate model fragments from metamodel footprint tuples*. The transformation of footprint tuples to model fragments is based on *partitioning the domain* of each property in the footprint and combining in every possible way tuples referring to the same operation. For partitioning we follow the approach presented in [13]. Attributes are partitioned depending on their types, associations are partitioned depending on their cardinality. For instance, Table II presents the partitions for the Class Diagram metamodel. The *is_primary* and *is_persistent* attributes are partitioned in two partitions True and False. The *name* attribute is partitioned in two partitions *name empty* and *name not empty*. The *parent* relation is partitioned by the number of associated *Classes*: 0 or 1. The *type*, *dest* and *src* relations have only one partition: 1 since their cardinalities are exactly 1. Finally *association* and *classifier* have three partitions: 0, 1, more than 1.

TABLE II
PARTITIONS FOR THE CLASS DIAGRAM METAMODEL

Metamodel feature	Partitions
Attribute::is_primary	true, false
Attribute::name	"", x | x!=""
Attribute::#type	1
Classifier::name	"", x | x!=""
Class::is_persistent	true, false
Class::#parent	0, 1
Class::#attrs	1, x | x>1
Association::name	"", x | x!=""
Association::#dest	1
Association::#src	1
ClassModel::#association	0, 1, x | x>1
ClassModel::#classifier	0, 1, x | x>1

The strategy we propose creates several model fragments per operation. Each model fragment of an operation contains the features of all the operation's tuples. For classes (e.g., *Classifier* and *ClassModel* for *getAllClasses* operation) the model fragment requires an instance of this class. When those features are classes' properties (*ClassModel::classifier* for *getAllClasses* operation) then we create several model fragments, one per partition. For instance, *getAllClasses* operation has 3 model fragments, each one requires a *Classifier* and a *ClassModel*, each one requires a different number of *classifier* following the partitioning of *classifier* property: no *classifier*, 1 *classifier*, more than one *classifier*. In the same way, *getPersistentClass* has 4 model fragments, they combine the different partitions of the properties *is_persistent* (True or False) and *parent* (no *parent* or one *parent*).

TABLE III
SUBSET OF MODEL FRAGMENTS GENERATED USING FOOTPRINT STRATEGY

Model-Fragment	Description
MFgetAllClasses1	A Classifier and a ClassModel cm | #cm.classifier = 0
MFgetAllClasses2	A Classifier and a ClassModel cm | #cm.classifier = 1
MFgetAllClasses3	A Classifier and a ClassModel cm | #cm.classifier > 1
MFgetPersistentClass1	A Class c|c.is_persistent=True and a Class c2|#c2.parent=0
MFgetPersistentClass2	A Class c|c.is_persistent=True and a Class c2|#c2.parent=1
MFgetPersistentClass3	A Class c|c.is_persistent=False and a Class c2|#c2.parent=0
MFgetPersistentClass4	A Class c|c.is_persistent=False and a Class c2|#c2.parent=1

The 7 model fragments generated from the operations *getAllClasses* and *getPersistentClass* are shown in Table III. Those model fragments should be contained in the test models we generate. For instance, in the fragment *MFgetAllClasses1* we require at least one *ClassModel* object cm which doesn't have any *classifier*. In another fragment *MFgetAllClasses2* we require cm to have only one *classifier*. In *MFgetAllClasses3* cm needs to have more than one classifier. The fragments of *getPersistentClass* operation cover all combinations of partition values for *is_persistent* attribute of a *Class* and for the number of *parents* of another *Class*.

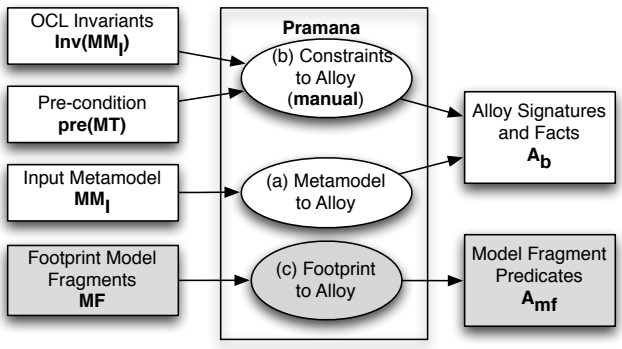


Fig. 4. Transformation to ALLOY

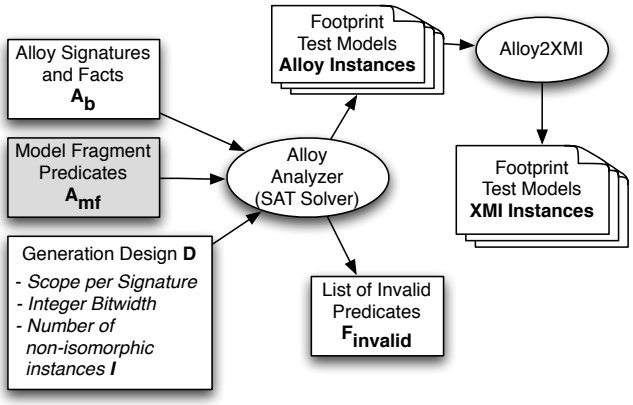


Fig. 5. Generation of test models

We generate a total of 72 model fragments representing combinations of partitions of features found in the footprints of class2rdbms. The footprint to model fragment transformation is depicted as step (2) of Figure 3.

IV. GENERATING TEST MODELS FROM TESTING KNOWLEDGE

In this section we present a methodology to generate test models from *footprint model fragments*. We describe its application on class2rdbms in the following subsections. Section IV-A describes the transformation of heterogeneous sources of knowledge to ALLOY (Figure 4). Section IV-B describes the generation of test inputs by solving the obtained ALLOY models (Figure 5).

A. Transformation to ALLOY

Our methodology to generate test models involves transformation of five sources of knowledge to the lightweight formal specification language ALLOY [17], [18]. The final formal ALLOY represents a *constraint satisfaction problem* that when solved gives us one or more test models. In Figure 4 we represent the four sources available and their target artifacts in ALLOY: (1) Input metamodel MM_I to ALLOY signatures and facts A_b, (2) OCL invariants $Inv(MM_I)$ and (3) pre-conditions $pre(MT)$ to ALLOY facts, (4) Footprint model fragments (from Figure 3) to ALLOY predicates.

(I) Metamodel MM_I to ALLOY. PRAMANA transforms a metamodel MM_I expressed in the EMF format Ecore using the transformation rules presented in [27] to ALLOY. Classes in the input metamodel are transformed to ALLOY signatures and implicit constraints such as inheritance, opposite properties, and multiplicity constraints are transformed to ALLOY facts in the base model A_b as shown in Figure 4.

(II) Constraints to ALLOY. We need to address the issue of transforming invariants and pre-conditions expressed on metamodels in the industry standard OCL to ALLOY. In the current version of PRAMANA, we *manually transform* OCL constraints to ALLOY facts in A_b as shown in Figure 4. The automatic transformation of OCL to ALLOY presents a number of challenges that are discussed in [1]. The core of ALLOY is declarative and is based on first-order relational logic with quantifiers while OCL includes higher-order logic and has imperative constructs to call operations and messages making some parts of OCL more expressive and difficult to transform to ALLOY in the most general case. In our case study, we have been successful in transforming all meta-constraints on the simplified UMLCD metamodel to ALLOY from their original OCL specifications.

(III) Footprint Model Fragments to ALLOY. We automatically transform model fragments to a set of ALLOY predicates A_{mf}. This transformation is purely syntactic. In our use case, the model fragments of Table III are translated into the predicates of Listing 1.

```
pred MFgetAllClasses1 {some Classifier and some cm:
    ClassModel | #cm.classifier=0}

pred MFgetAllClasses2 {some Classifier and some cm:
    ClassModel | #cm.classifier=1}

pred MFgetAllClasses3 {some Classifier and some cm:
    ClassModel | #cm.classifier>1}

pred MFgetPersistentClass1 {some c:Class, c2:Class| c.
    is_persistent = True and #c2.parent = 0}

pred MFgetPersistentClass2 {some c:Class, c2:Class| c.
    is_persistent = True and #c2.parent = 1}

pred MFgetPersistentClass3 {some c:Class, c2:Class| c.
    is_persistent = False and #c2.parent = 0}

pred MFgetPersistentClass4 {some c:Class, c2:Class| c.
    is_persistent = False and #c2.parent = 1}
```

Listing 1. Footprint as Alloy Predicates Representing Combination of Partitions

The model fragment to ALLOY predicate transformation is a new contribution, integrated into the PRAMANA tool chain. This is depicted in grey boxes of Figure 4.

B. Generating Test Models

The ALLOY signatures, facts, and predicates (A_b and A_{mf}) are transformed to a set of expressions in relational calculus by the ALLOY analyzer. These expressions are then transformed to Conjunctive Normal Form (CNF) using KodKod [32]. Finally, the CNF is solved using a SAT solver [23].

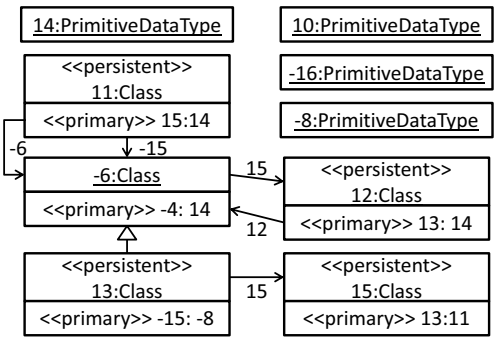


Fig. 6. Generated Test Model from one Model Fragment

The low-level SAT solutions are transformed back to XMI[4] models that conform the initial metamodel MM_I, $pre(MT)$, $Inv(MM_I)$, and the predicates in A_{mf}. In Figure 6, we present a test model generated by solving the predicate for fragment MFgetPersistentClass2, defined in Listing 1.

A certain number of predicates in A_{mf} may not conform to the base ALLOY model A_b. An attempt to generate a test model by executing a run command for such predicates fails with no instance. This implies that a model fragment predicate is inconsistent with respect to the *input domain specification* represented by A_b. The invalid predicates are inconsistent with respect to ALLOY signatures and facts from either MM_I, $Inv(MM_I)$, $pre(MT)$, or a conjunction of facts. For instance, MFgetAllClasses1 is inconsistent with an Alloy fact specifying that a Classifier is contained by a ClassModel, as illustrated in the MM, Figure 2(a). We weed out these inconsistent predicates into a set $F_{invalid}$. Predicates in $F_{invalid}$, are removed from A_{mf} for further generation. From the original set of 72 model fragment predicates generated for class2rdbms, 23 are consistent and can be used for test generation.

Starting a test model generation in ALLOY requires ALLOY run statements that depict the number of objects (atoms in ALLOY) that need to be generated for each class (signature in ALLOY) in order to solve a predicate. The run statements also must specify bounds/scopes on Integer values and sequence lengths. These signatures can be seen as *factors* and their scopes as *factor levels* in the experimental design [11] [15] parlance. A generation design D is the fifth source of knowledge (Figure 5). It is a set of ALLOY run statements that contain the bounds/scope for each ALLOY signature and Integer ranges for each of the M predicates, where M is the number of fragment predicates in A_{mf}. The set of run statements in D are generated from a set of numerical parameters. For instance, in Listing 2, we present a run command to solve predicate MFgetPersistentClass2 with the scope for each signature. The *exactly* keyword ensures the exact number of objects of a certain type in the model.

```
run MFgetPersisentClass2 for 1 ClassModel,5 int,exactly 10
    Class,exactly 5 Attribute,exactly 4 PrimitiveDataType,
    exactly 10 Association
```

Listing 2. Run Command to Generate Complete Model from a Fragment

A parameter to the ALLOY analyzer is the number of non-isomorphic test models that need to be generated per predicate. This parameter I is provided at the time of generation. Generating multiple non-isomorphic models allow us to increase the diversity of structure in the test models generated. The generation of non-isomorphic models is based on the *symmetry breaking scheme* of ALLOY described in [16]. The symmetry breaking scheme adds a Boolean constraint to break the symmetry between the *current test model* and the *next test model* such that every test model is non-isomorphic with respect to the previous one. The number of symmetry breaking constraints depends on the number of test models required from the ALLOY analyzer. In this paper, we validate the effectiveness of our approach by generating multiple non-isomorphic models for the same model fragment predicate.

V. EXPERIMENTS

In this section, we perform experiments to address **Challenge 3**: *Are automatically completed models from transformation knowledge effective in detecting faults in the transformation?*

The experimentation is divided into two steps:

1) We generate several sets of test models following our white-box methodology and two black-box methodologies from related work.
2) We perform *mutation analysis* [25] to evaluate and compare the fault detecting effectiveness of the generated test sets.

When using our methodology we generate test models for class2rdbms as described in previous sections. We produce 10 non-isomorphic solutions for each one of the 23 predicates obtaining a test set of 230 test models. These models and other software artifacts are available on the experimentation website[5].

We replicate the generation process 8 times obtaining 8 sets of test models. Each set is generated using different design parameters, with increasing values, as summarized in Table IV. For each set we ask ALLOY to find models made of a different number of ClassModels, Classes, Associations, Attributes, PrimitiveDataTypes, and we give a specific range for the integer values of the properties.

[4]http://www.omg.org/spec/XMI/2.4.1/

[5]https://sites.google.com/site/staticfootprinting/

TABLE IV
GENERATION DESIGN PARAMETERS FOR TEST MODEL GENERATION

Factors: **Sets:**	1	2	3	4	5	6	7	8
#ClassModel	1	1	1	1	1	1	1	1
#Class	5	5	10	10	5	10	5	10
#Association	5	10	5	10	5	5	10	10
#Attribute	25	25	25	25	30	30	30	30
#PrimitiveDataType	4	4	4	4	4	4	4	4
Bit-width Integer	5	5	5	5	5	5	5	5
#predicates	23	23	23	23	23	23	23	23
#models/predicates	10	10	10	10	10	10	10	10

A. Injecting Faults in the Model Transformation

We evaluate the generated test sets based on the principles of mutation analysis [9]. Mutation analysis involves creating a set of faulty versions or *mutants* of a program (i.e., model transformation in our case). A test set must distinguish the correct program output from the output of all its mutants. In practice, faults are modelled as a set of mutation operators, each one representing a class of faults. A mutation operator is applied to the program under test to create every possible mutant with a single injected fault. A mutant is killed when at least one test model detects the pre-injected fault, i.e., when program output and mutant output are different. A test set is relatively adequate if it kills most mutants of the original program. A mutation score is associated to the test set to measure its effectiveness in terms of percentage of the killed (i.e., revealed) mutants.

We inject faults in a transformation using mutation operators for model transformations presented by Mottu et al. [25]. These mutation operators are based on the three basic operations of a model transformation: navigation of the input models through relations between classes, filtering of collections of objects, creation and modification of the elements of the output model. For instance, by applying the mutation operator *Relation to the same class change (RSCC)*, the navigation of one association toward a class is replaced with the (erroneous) navigation of another association to the same class. Using this approach the following mutation operators were defined in [25]:

- Relation to the same class change (RSCC)
- Relation to another class change (ROCC)
- Relation sequence modification with addition (RSMA)
- Relation sequence modification with deletion (RSMD)
- Collection filtering change with perturbation (CFCP)
- Collection filtering change with addition (CFCA)
- Collection filtering change with deletion (CFCD)
- Class compatible creation replacement (CCCR)
- Classes association creation addition (CACA)
- Classes association creation deletion (CACD)

We apply these operators on class2rdbms. We identify all the possible matches of patterns described by each operator. For each match we generate a new mutant of class2rdbms. The different mutation types contribute to a total of 200 mutant transformations for class2rdbms, as shown in Table V.

In general, not all injected mutations become faults, since some mutants may have semantics equivalent to the correct program, and therefore are undetectable. The controlled experiments presented in this paper use mutants presented in our previous work [25] where we have already identified and filtered equivalent mutants.

B. Results and Discussion

In this section, we present a number of observations resulting from our experiments.

1) ***Q1:*** *Does static analysis of a transformation give effective knowledge to generate test models?:* We illustrate in grey within the box-plot of the Figure 7 the mutation scores for each footprint test set. Most of the sets (5 among 8) get 98.97% mutation score. The score of 98.97% means that only 2 mutants remain alive (192 killed mutants /194 non-equivalent mutants). The average is 97.62%, meaning that less than 5 mutants remain alive on average. Those high scores reveal that the proposed strategy returns consistently good test sets.

We compare our footprint strategy with an unguided strategy. The unguided strategy only uses knowledge from the input domain and no special testing knowledge. The two box-plots at the right of the Figure 7 represent the distribution of the mutation score of test model sets generated without the use of testing knowledge. We generate two sets of unguided test models : 90 models/set in 8 sets and 180 models/set in 8 sets using the ALLOY Signatures and Facts A_b. Each test model is non-isomorphic with respect to the others. The two sets were already been used in [28] and they are only different in the number of non-isomorphic test models. The mutation score of these unguided models is around 70%. This is more than 23% lower than the minimum mutation score with our footprint strategy and about 29% less than the maximum. There is not much variation in mutation score for models generated using the unguided strategy despite the use of non-isomorphic test models clearly indicating the need for testing knowledge (black- or white-box).

We also compare our footprint strategy to input domain partitioning. In Figure 7, we present 2 boxes representing mutation scores for test sets based on input domain partitioning. The testing strategy $AllPartitions$ specifies that all partitions of each metamodel property must be covered by a test model. $AllRanges$ specifies that each range in each property partition must be covered by a different test model, requiring several models for covering a partition. They are black-box strategies since they use only the input domain of the transformation. As shown in Figure 7, the AllRanges strategy has an average of 89.9% and AllPartitions 87.5%. The experimentation we perform on black-box input domain partitioning strategies is analogous to the one we published in [28]. [6] Here, we are interested in the comparison of the three box-plots at the left of the Figure 7. We observe a significant increase in the mutation scores we obtain. The entire box for the footprint strategy is superior to all the other boxes (each box representing the results between the first and the third quartile). Even the minimum score obtained with footprints is higher than the black-box maximum scores.

2) ***Q2:*** *What is the effect of generating multiple non-isomorphic models?:* In [28], we concluded by saying that AllRanges and AllPartitions strategies are more efficient than unguided strategy with both median scores around 82%. In this paper, we take a step further to generate 10 models per predicate. This is due to our new setup for mutation analysis on a grid. Therefore we can take advantage of the non-isomorphic

[6] However here we generate 10 extra non-isomorphic models per predicate while in [28] we had only one per predicate. For this reason our current mutation score is higher than [28], where we obtained average scores around 82%.

TABLE V
NUMBER OF CLASS2RDBMS MUTANTS PER MUTATION OPERATOR

Mut. Oper.	CFCA	CFCD	CFCP	CACD	CACA	RSMA	RSMD	ROCC	RSCC	Total
# of Mutants	19	18	38	11	9	72	12	12	9	200

criteria of our generation.

We observe an increase in scores for AllRanges and AllPartitions from [28] to our results in Figure 7. A gain of 6% and 11% is observed. We also notice that while the box-plots for *unguided strategies* were dispersed without non-isomorphic criteria (6% and 3% between the first and third quartiles, 13% and 3.7% between the maximum and minimum scores), here they converge to 70.1%. We conclude that non-isomorphic criteria reduces the variability and increases the efficiency of the test model generation made with PRAMANA.

3) ***Q3:*** *How to deal with remaining live mutants?:* In Table VI, we enlist live mutants per set. *f*, *n* letters are the initials of *filtering* and *navigation* indicating the type of operation altered with the mutation.

The influence of the generation design parameters on the number of live mutants looks minor, as we can see with the spread of the grey box-plot: 2.2% between the first and third quartiles. The difference between the design parameters of each set is not significant to conclude a correlation between design parameters and live mutants.

From Table VI, we observe that 2 mutants remain alive with all the sets: *f19*, *f21*. Clearly, we lack testing knowledge to kill them. The rest of the mutants are killed because the footprint strategy gives test models with relevant knowledge.

Analyzing the live mutants we notice that they are affected by the mutation operator *CFCA: A Collection is Filtered Additionally*. For instance, the injected fault *subSequence(0,0)* in the following expression is an additional filter.

```
getAllClasses(model).select{ c | c.parent == cls }
                    .subSequence(0,0) //Injected fault
```

Here, cls is a Class selected before (expression not shown), c is any Class of the model. The instruction selects the classes which inherit from cls. The mutants wrongly (intentionally injected) select only the first child Class of cls. This mutant

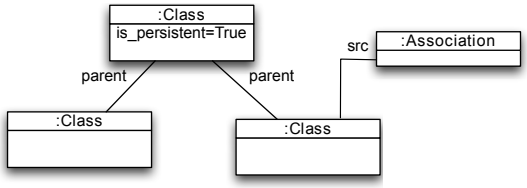


Fig. 8. Human testing knowledge to kill live mutants

TABLE VI
LIVE MUTANTS AFTER MUTATION ANALYSIS

Set	**#live mutants**	**Live mutants**
1	2	f19, f21
2	2	f19, f21
3	6	f19, f21, f10, f31, f48
4	2	f19, f21
5	2	f19, f21
6	12	f19, f21, f10, f31, f73, n99 to n105
7	10	f19, f21, f70, n92 to n98
8	2	f19, f21

expression appears in two operations of class2rdbms. One that collects attributes (in *f19*) and the other collects associations (in *f21*). The only way to see a difference in output and detect these mutants is to have at least two child Classes with one Association and Attribute as shown in Figure 8.

Our strategy cannot kill these mutations since the inheritance relationship in the UMLCD metamodel is not bidirectional, as illustrated Figure 2. Only the *parent* property exists and not *child*. A partition of a *child* property could have resulted in the creation of more than one child classes in test models. We are evaluating how to extend our footprint strategy to introduce a partitioning of implicit opposite references. However, with the current tool, a user can still add this partial knowledge describing it in a *partial test model* as discussed in our work [31].

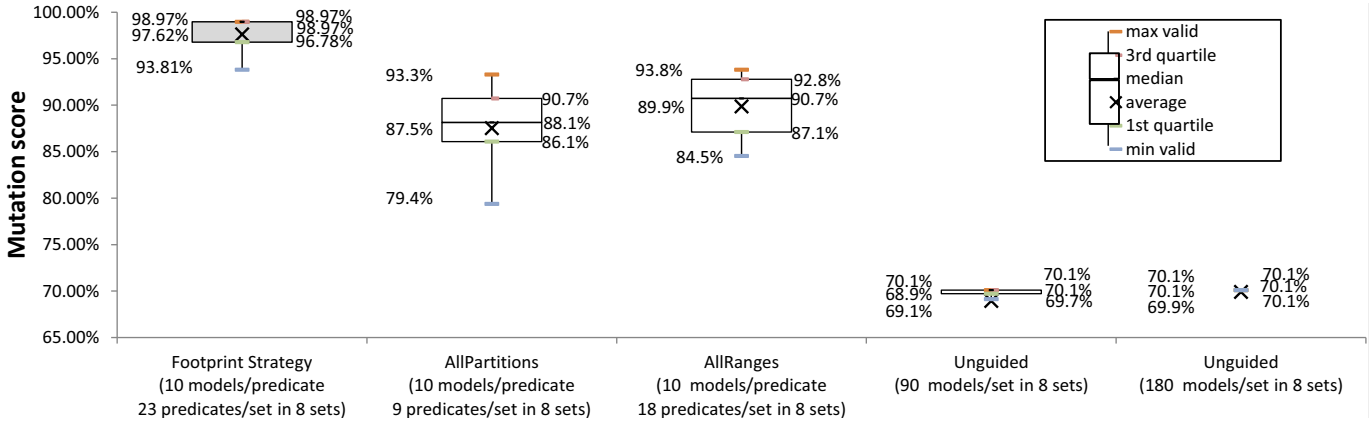


Fig. 7. Mutation scores for footprint predicates

4) ***Q4:*** *Are generated test models time efficient in the proposed methodology? :* We use a Macbook Pro Intel dual-core processor 2.7 GHz, 8 GB RAM to generate test models and a grid of 16 desktop Intel dual-core processor 2.4 GHz, 3 GB RAM to perform mutation analysis. We measure the time to generate the test model sets. They are summarized in Figure 9. The slowest generation is 20 seconds for Set 8 while the fastest is 4.6s for Set 1. This shows that concise test models can be produced in a reasonable time by our footprinting method. There is a correlation between the generation design parameters and the time to generate a set. The more model objects are requested to the solver, the longer is the time to generate the test models. As we cannot strictly correlate the size of the models and the fault detecting effectiveness of a set, we cannot correlate time of generation and the fault detecting effectiveness.

We also evaluate the effort a tester will have to make to use the proposed methodology. First, she translates the OCL constraints (i.e., metamodel invariants and transformation pre-conditions) as discussed in Section IV-A. Second, she creates a generation design: she chooses generation parameters to fill a column of Table IV and generates ALLOY run statements. Then the tester can automatically run the test set generation, choosing a number of non-isomorphic models per predicate. Once the automatic generation is performed, the tester can model his additional testing knowledge in the form of a partial model. We evaluate in [31] that writing such a partial model and automatically completing them using our methodology is less tedious than writing complete test models by hand.

C. Threats to Validity

Our approach can scale to a relatively large input domain such as the full UML with 246 classes and 546 properties by using strategies such as metamodel pruning [30]. Pruning extracts a concise pruned metamodel with fewer classes and properties that type-matches the original large metamodel. The type conformance ensures that the pruned metamodel instances are also instances of the original large metamodel. Pruning uses a set of required classes and properties as input, detects its minimal obligatory dependencies to produce the pruned metamodel. The pruned metamodel can be transformed to an ALLOY model that is tractable by a SAT solver.

The results of these experiments are based on statically analyzing a representation of the transformation in Kermeta [24]. Kermeta is an imperative transformation language similar to Java. At this point of the work, we cannot yet claim that the approach will be equally effective for rule-based transformation systems such as ATL [21] or graph transformation based such as VIATRA [33].

The models generated contain integers in place of strings for properties of type String in the metamodel. This simplification allows ALLOY to focus on solving for model structure rather than string content. Some transformations may be heavily dependent on string structure of properties. We do not address such a scenario since it occurs rarely in model transformations we are aware of.

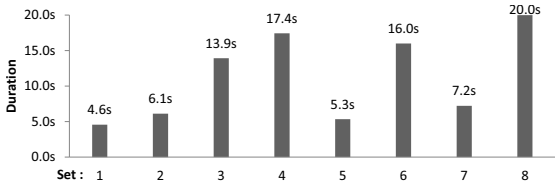


Fig. 9. Time to generate test sets

The combined effect of a testing strategy and built-in generation of multiple non-isomorphic solutions in ALLOY to give high mutation scores has not been well studied in our experiments. In future, we want to find ways of clearly discerning the contribution of strategies and their interaction with symmetry breaking schemes [16] in ALLOY.

VI. RELATED WORK

Our work is related to two main areas: generation of (test) models, and strategies for guiding test generation.

Several proposals deal with model generation for applications such as testing. Model generation is more general and complex than generating integers, floats, strings, lists, or other standard data structures such as those dealt with in the Korat tool of Chandra et al. [5]. Korat is faster than ALLOY in generating data structures such as binary trees, lists, and heap arrays from the Java Collections Framework but it does not consider the general case of models which are arbitrarily constrained graphs of objects. Constraints on models make model generation a different problem than generating test suites for context-free grammar-based software [14] which do not contain domain-specific constraints. In [6], the authors present an automated generation technique for models that conform only to the class diagram of a metamodel specification. A similar methodology using graph transformation rules is presented in [10]. Generated models in both these approaches do not satisfy the constraints on the metamodel. In [29], we present a method to generate models given partial models by transforming the metamodel and partial model to a Constraint Logic Programming (CLP). We solve the resulting CLP to give model(s) that conform to the input domain. However, the approach does not add new structural elements to the model. The authors initialize the number of structural elements such as objects and relations in the partial model and hope that it is sufficient for obtaining complete test models. The constraints in this system are limited to first-order horn clause logic. In [27] Sen et al., we address the issue of generating test models that satisfy constraints on both structure and properties. We presented the tool CARTIER (now called PRAMANA) based on the constraint solving system ALLOY to generate test models by combining heterogenous sources of knowledge. These sources include the input metamodel, invariants on it, pre-conditions, and testing strategies. A similar tool UML2Alloy [2] takes as input UML class models with OCL constraints to generate an ALLOY model for analysis.

Test model generation need to be guided using strategies. In [12], the authors present *input domain partitioning strategies*

that are used to generate model fragments. It is only in our paper [28], that we use these strategies to generate test models for black-box testing a transformation. Automating white-box testing for a model transformation requires the use of knowledge in a transformation to generate test models. In [31], we *manually extract* such knowledge in the form of partial models to generate highly effective test models. In this paper, we go a step further and use static analysis to extract testing knowledge from a transformation. Not much work has been published about white-box model transformation testing and the only directly related reference we are aware of is [22] that proposes a set of three criteria to guide the creation of tests for model transformations. With respect to our work, the methodology in [22] has a low degree of automation, does not consider generic transformation invariants and pre-conditions, and is not supported by a quantified experimental evaluation.

VII. Conclusion

In this paper we presented a tool-supported methodology to automatically generate test cases using structural information from a model transformation. The methodology makes use of the metamodel footprinting mechanism, generates partial models representing the testing intent and uses the ALLOY solver to create complete usable models. The experimental results show that a limited amount of white-box information on the model transformation (i.e., our footprints) can provide remarkable improvements on the efficiency of the generated tests. The paper can be also interpreted as evidence of the fact that the highly structured nature of model-transformation languages makes them particularly suitable for automated testing.

In future work we plan to 1) extend the experimentation to industrial testing scenarios; 2) deepen the static analysis phase to extract partial models representing complex input patterns; 3) apply our methodology to other model transformation languages (e.g. ATL, QVT), to quantify how the improvement in testing efficiency is dependent on the model-transformation language.

References

[1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. Uml2alloy: A challenging model transformation. In *MoDELS*, pages 436–450, 2007.

[2] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On Challenges of Model Transformation from UML to Alloy. *Software and Systems Modeling, Special Issue on MoDELS 2007*, 9(1):69–86, 2008.

[3] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu. Barriers to Systematic Model Transformation Testing. *Communications of the ACM*, 53(6), 2010.

[4] J. Bezivin, B. Rumpe, A. Schurr, and L. Tratt. Model transformations in practice workshop, october 3rd 2005, part of models 2005. In *Proceedings of MoDELS*, 2005.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, 2002.

[6] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *Proceedings of ISSRE'06*, Raleigh, NC, USA, 2006.

[7] F. Budinsky. *Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2004.

[8] H. Czarnecki. Feature-based survey of model transformation approaches. *IBM Systems Journal.*, 2006.

[9] R. DeMillo, R. J. Lipton, and F. Sayward. Hints on test data selection : Help for the practicing programmer. *IEEE Computer*, 11(4):34 – 41, 1978.

[10] K. Ehrig, J. Kuster, G. Taentzer, and J. Winkelmann. Generating instance models from meta models. In *FMOODS*, pages 156 – 170., Bologna, Italy, June 2006.

[11] W. T. Federer. *Experimental Design: Theory and Applications*. Macmillan, 1955.

[12] F. Fleurey, B. Baudry, P.-A. Muller, and Y. L. Traon. Towards dependable model transformations: Qualifying input test data. *Software and Systems Modelling*, 2007.

[13] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jezequel. Model-driven engineering for software migration in a large industrial context. In *MoDELS/UML*, 2007.

[14] M. Hennessy and J. Power. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *Proc. of the 20th IEEE/ACM ASE*, NY, USA, 2005.

[15] D. Hoskins, R. C. Turban, and C. J. Colbourn. Experimental designs in software engineering: d-optimal designs and covering arrays. In *WISER'04*, pages 55–66, New Port Beach, CA, USA, 2004.

[16] S. Ilya. Generating effective symmetry-breaking predicates for search problems. *Discrete Appl. Math.*, 155(12):1539–1548, June 2007.

[17] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, March 2006.

[18] D. Jackson. http://alloy.mit.edu. 2008.

[19] C. Jeanneret, M. Glinz, and B. Baudry. Estimating footprints of model operations. In *International Conference on Software Engineering (ICSE'11)*, Honolulu, USA, May 2011. IEEE.

[20] C. Jeanneret, M. Glinz, and B. Baudry. Footprinting operations written in kermeta. technical report ifi-2011.0002. Technical report, University of Zurich, 2011.

[21] F. Jouault and I. Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 128–138. Springer Berlin / Heidelberg, 2006.

[22] J. Kuster and M. Abd-El-Razik. Validation of model transformations - first experiences using a white box approach. In *MoDeVa associated to MoDELS*, Genova, Italy, October 2006.

[23] Y. S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient sat solver. In *Lecture Notes in Computer Science SAT 2004 Special Volume LNCS 3542.*, pages 360–375, 2004.

[24] N. Moha, S. Sen, C. Faucher, O. Barais, and J.-M. Jezequel. Evaluation of kermeta for solving graph-based problems. *STTT*, 12(3-4):273–285, 2010.

[25] J.-M. Mottu, B. Baudry, and Y. Le Traon. Mutation analysis testing for model transformations. In *Proceedings of ECMDA'06*, Bilbao, Spain, July 2006.

[26] OMG. The Object Constraint Language Specification 2.0, OMG: ad/03-01-07, 2007.

[27] S. Sen, B. Baudry, and J.-M. Mottu. On combining multi-formalism knowledge to select test models for model transformation testing. In *ICST*, Lillehammer, Norway, April 2008.

[28] S. Sen, B. Baudry, and J.-M. Mottu. Automatic model generation strategies for model transformation testing. In *ICMT*, pages 148–164, 2009.

[29] S. Sen, B. Baudry, and D. Precup. Partial model completion in model driven engineering using constraint logic programming. In *International Conference on the Applications of Declarative Programming*, 2007.

[30] S. Sen, N. Moha, B. Baudry, and J.-M. Jezequel. Meta-model pruning. In *Model Driven Engineering Languages and Systems, 12th International Conference (MODELS)*, Denver, CO, USA, October 4-9 2009.

[31] S. Sen, J.-M. Mottu, M. Tisi, and J. Cabot. Using models of partial knowledge to test model transformations. In *ICMT*, Prague, Czech Republic, May 2012.

[32] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for Construction and Analysis of Systems*, Braga,Portugal, March 2007.

[33] D. Varro and A. Balogh. The model transformation language of the viatra2 framework. *Science of Computer Programming*, 68(3):214–234, 2007.